ELSEVIER

Theoretical Computer Science 160 (1996) 217–239

Theoretical Computer Science

CGOOD, a categorical graph-oriented object data model

Chris Tuijn [a,*], Marc Gyssens [b,1]

[a] *Agfa-Gevaert N.V., B-2640 Mortsel, Belgium*
[b] *University of Limburg (LUC), B-3590 Diepenbeek, Belgium*

Received August 1993; revised August 1994
Communicated by M. Nivat

Abstract

While the relational data model and many of its extensions have proven to be of considerable importance to many database applications, it has become clear that some advanced systems require more flexible structures and query languages. The expression of queries based on the occurrence of substructures on instance level (i.e., pattern matchings) requires constructs which cannot be expressed easily in the traditional models.

In this article, we introduce an object-oriented data model which solves these shortcomings. The instances of this data model will be represented by typed graphs. Both scheme and data will be defined entirely in terms of categorical constructs; pattern matching of graphs will be realized by morphisms in a suitable graph category. These morphisms will be used to define a powerful query and update language, which is capable of querying and restructuring the database in a natural and elegant way. Finally, we show that this query language is able to express the relational database operators, functional abstraction and transitive closure.

It will become clear that the categorical approach provides a solid basis for data modeling because it offers a unifying, theoretical framework. The abstractive power of the categorical framework creates an environment which sheds new light upon existing concepts and is the source of many interesting generalizations. The capability to make abstraction of low-level details, moreover, will often simplify the proofs of many theorems which would be rather involved and confusing in the traditional framework3.

0. Introduction

The introduction of the relational database model [7, 15] two decades ago has been of crucial importance for both research and development in the database world. What has

* Corresponding author. E-mail: chris@eps.agfa.be.

[1] This paper presents research results of the Belgian Incentive Program "Information Technology" – Computer Science of the future, initiated by the Belgian State – Prime Minister's Service – Science Policy Office. The scientific responsability is assumed by its authors.

SSDI 0304-3975(95)00089-5

made the relational approach so appealing was the combination of both mathematical simplicity and expressive power.

Since then, a considerable amount of new formalisms has been introduced for various reasons. One of these reasons was the desire to also be able to express hierarchical structures. Therefore, the relational model was extended to support so-called nested relations [14, 17]. The general trend of putting more structure into the models became obvious by the growing interest in semantic network models [12], complex-object models [3] and, later, in the object-oriented data models [1, 9, 21].

The need to incorporate more structure into the scheme and instances of the data models, however, has made the definition and manipulation of the data in these models more complicated. Therefore, an easy, graphical representation of the data seems to be an absolute requirement. In this context, the use of graphs seems to be the ideal choice to represent the data. In order to express powerful queries (based on pattern matching of graphs), graphs must also be part of the query language. This approach has been taken, e.g., in the GOOD data model [9], where queries can also be visualized as graphs.

In an earlier paper, we presented an object-oriented database model (called the *typegraph model*) which was entirely based on categorical notions [21]. The use of category theory [2, 6, 11] as the basis for data models offers both the advantage of having a theoretical framework to work in and a graphical representation of the data. An additional feature of category theory is that it allows to define operations in a very uniform way by using so-called universal properties, a way to define concepts which has been proven to be the source of many interesting applications in category theory. Category theory also provides an excellent framework to make abstraction of certain details. As such, objects can be specified in terms of their relationship to other objects instead of their explicit low-level description. In this article, we introduce CGOOD (a Categorical Graph-Oriented Object Database Model). In this model, the sophisticated pattern-based query capabilities of GOOD are combined with the categorical techniques introduced in the typegraph model. This is realized by expressing the pattern matching of graphs by morphisms in a suitable graph category. We thus will obtain a simple, robust yet very powerful query language which is solely based on categorical constructs.

This article is organized as follows. In Section 1, a formal definition of the CGOOD data model is given. Instances are defined as typed graphs within a graph category. The morphisms in this category define pattern matching of graphs in a straightforward way. In Section 2, we introduce the query language of CGOOD. Both the *addition* and *deletion* of information to a given instance will be defined using a *pattern* and a graph, denoting how the pattern must be transformed. The query operation is defined by a universal property construction. In Section 3, the language is extended with recursive power by means of a fixpoint construction. In Section 4, we give a few examples of the query language to show the expressive capability of the data model. In particular, we study the expressibility of the relational database operators, the abstraction operators and the transitive closure.

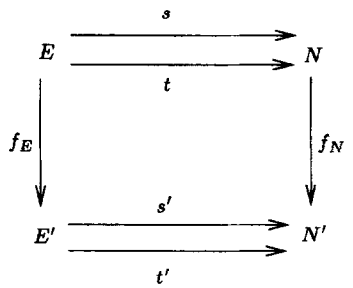
1. The CGOOD model

The data in CGOOD will be represented using unlabeled graphs. The objects of an instance will be defined by a set N of nodes and their properties by a set E of edges. Between E and N two total functions s and t will define both source and target of an edge. An unlabeled, directed graph can thus be characterized by a quadruple (E,N,s,t). The collection of these quadruples defines a category $\mathcal{G}$.

Definition 1.1. The category $\mathcal{G}(Obj, Arr, Dom, Codom, \circ)$ of unlabeled graphs is defined as follows:

(1) *Obj* is defined by the quadruples (E,N,s,t) where E and N are sets and s and t total functions from E to N.

(2) An object f in *Arr* from (E,N,s,t) to (E',N',s',t') is defined as a pair (f_E, f_N) of total functions $f_E : E \to E'$ and $f_N : N \to N'$ such that $s' \circ f_E = f_N \circ s$ and $t' \circ f_E = f_N \circ t$, as indicated in the figure below.

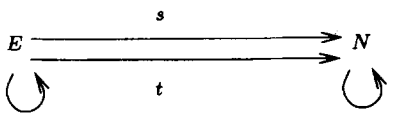


If G is an object of the category $\mathcal{G}$, we will often denote the associated quadruple as (E_G, N_G, s_G, t_G).

Almost all constructions in this article will be executed within the category $\mathcal{G}$. Therefore, it is important to investigate which constructions are well-defined. In particular, we are interested in the availability of cartesian products, limits and colimits (i.e., operators which all are defined in categories called *topoi*). We therefore formulate following lemma:

Lemma 1.1. *The category $\mathcal{G}$ is a topos.*

Proof. The category $\mathcal{G}$ can be written as $\mathbf{Set}^{\mathcal{D}}$, the category of functors between $\mathcal{D}$ and **Set** where $\mathcal{D}$ is defined by the following category:



Then, according to [4], $\mathcal{G}$ is a topos. □

The category $\mathcal{G}$ provides the framework for what we will call *abstract instances*. These abstract instances will be defined as *typed* instances or, more explicitly, as morphisms in the category $\mathcal{G}$ between an unlabeled graph G and a unlabeled graph T. The unlabeled graph T will define the scheme of the database; in this graph, every node defines a type and every arrow a property.

Definition 1.2. Let G and T be elements of $Obj(\mathcal{G})$.

(1) An *abstract instance* defined by G over T is a morphism $g_T : G \to T$ in $\mathcal{G}$.

(2) The category $\mathcal{G}_T$ of typed graphs over T is defined as follows :

- $Obj(\mathcal{G}_T)$ is the collection of all abstract instances G over T with G in $\mathcal{G}$.
- A morphism in $\mathcal{G}_T$ between an object $g_T : G \to T$ and an object $g'_T : G' \to T$ is defined by a morphism $f : G \to G'$ in $\mathcal{G}$ such that $g_T = g'_T \circ f$.

(3) Let G'_T be another object in $\mathcal{G}_T$; if there exists a monomorphism $i : G'_T \hookrightarrow G_T$, we call G'_T a *subinstance* of G_T.

The definition of the category $\mathcal{G}_T$ is an example of what in category theory is known as a *slice*-category [5].

From now on, we will frequently use the notation $(\mathcal{G}_T, I_T)$ (or $(\mathcal{G}_T, I)$) where I_T (or I) is an abstract database instance over the database scheme T.

It should be noted that modeling incomplete information on instance level poses no problem because an *undefined attribute* of an object just has to be omitted from the set of edges E. This is an immediate consequence of the fact that instances are defined by mappings from the instance graph to the scheme graph. In many other models (such as, e.g., the typegraph model [21] and the sketch-based data models [4]) instances are functors from a scheme graph to the instance category of sets with total functions. Although this approach seems less appropriate here, it has some advantages with respect to query formulation (in terms of universal constructions). In this type of models, it also is easier to distinguish between functional and multivalued properties. For details, we refer to [18, 21].

The category $\mathcal{G}_T$ is also a topos and consequently supports the construction of finite limits and colimits.

Lemma 1.2. *Let T be a typegraph in $\mathcal{G}$. The category $\mathcal{G}_T$ of all abstract instances over T is a topos.*

Proof. In [4], it is shown that a slice category defined on a topos over an object is itself a topos. Since $\mathcal{G}_T$ can be considered as the slice category of $\mathcal{G}$ (which is a topos) over T, $\mathcal{G}_T$ is also a topos. □

The morphisms in the category $\mathcal{G}_T$ will be called *embeddings*, in analogy with [9]. It can be shown that the notion of embedding defined there corresponds to our definition. Embeddings define in a natural way the notion of *pattern matching*. For our purpose, it will be important to consider all embeddings from a pattern to an instance since we will want to apply modifications to a given instance for each embedding. That such

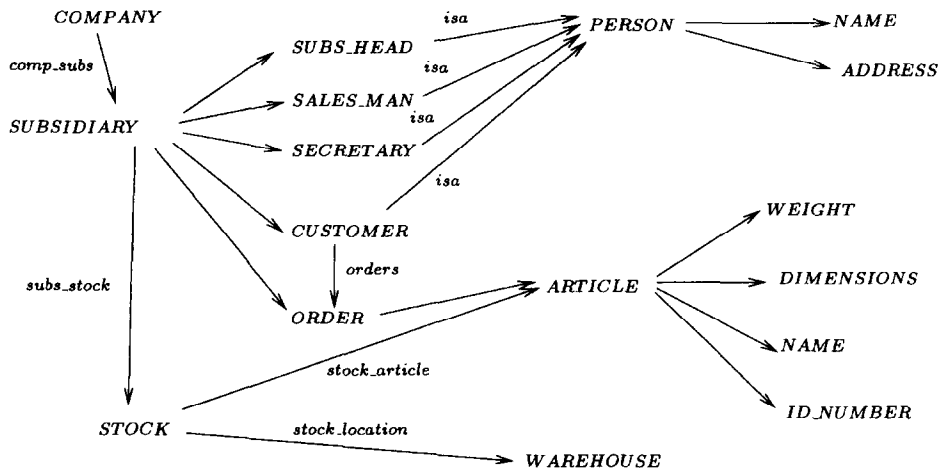


Fig. 1.

transformations are well-defined is already suggested by the fact that $\mathcal{G}_T$ is a topos and therefore supports the construction of the exponential which usually is related to the **Hom**-sets. It can be shown that although the exponential in $\mathcal{G}_T$ cannot be identified with a **Hom**-set construction, it nevertheless is closely tied with it [5].

Let us now consider the example of a *COMPANY* which has a number of *SUBSIDIARIES*. Each subsidiary has a number of employees which are either *SUBS_HEAD*, *SECRETARY* or *SALES_MAN*. Each subsidiary has a number of *CUSTOMERS* which have a number of *ORDERS* of *ARTICLES* pending. Each subsidiary also has a number of *WAREHOUSES* where the *STOCK* resides. The graph defining the scheme is shown in Fig. 1.

Instances in CGOOD are graph morphisms from a graph to the typegraph. This approach provides us immediately with a typed data model, because there is an immediate correspondence between each node (edge) in an instance graph with a specific node (edge) in the typegraph. The graph morphisms in the category of typed graphs (over a specific type T) allow us to define subgraphs or matching graphs within a fixed graph.

In CGOOD, we work with unlabeled graphs. This means that the typegraph is unlabeled too. To identify the edges and nodes of the typegraph, we label them with strings (upper case for types, lower case for attributes). We only do this to be able to formulate statements; from a categorical point of view, however, this is not necessary.

The example above shows that the definition of the data in CGOOD is very similar to GOOD [9]; it should be pointed out that, until now, we have not been distinguishing between functional and nonfunctional edges. In order to support these concepts, we will have to introduce new categories. These categories will become subcategories of $\mathcal{G}_T$; in these new categories, a number of constraints will be put on the source and target mappings. This implies that these new categories no longer have to be topoi. Therefore, for the time being, we prefer to keep working in $\mathcal{G}_T$. This will enable us to execute a number of constructions without having to worry about existence problems.

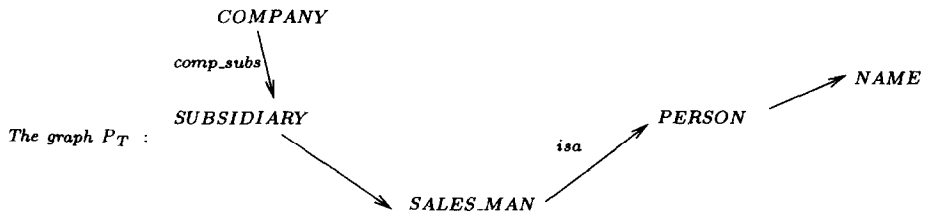


Fig. 2.

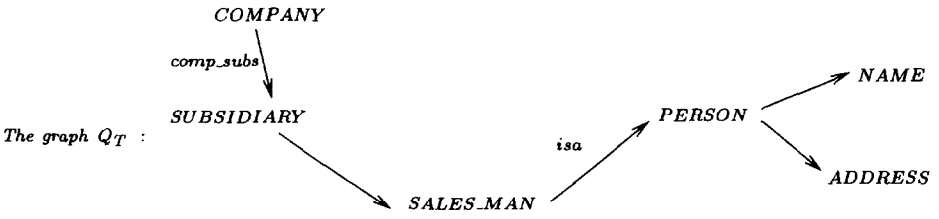


Fig. 3.

2. The CGOOD update/query language

2.1. Addition of information

To add information to a graph instance (i.e., an object of a given category $\mathcal{G}_T$), we must proceed in several steps. First, we have to specify which information we are going to add; then, we must describe how and where we are going to add this information. If, e.g., in the company example given above, we would like to add the address of a salesman of a given subsidiary, we first would define a subgraph P_T as in Fig. 2.

Next, we define a graph morphism f from this graph P_T to the database instance I_T. We now have established a link between pattern P_T and the actual subgraph of I_T we are going to update. The last phase consists of specifying what information we are going to add. This is established by extending the graph P_T to a graph Q_T. In our example, we define a monomorphism i from P_T to Q_T as Fig. 3.

We now have to *add* the information together. This can be done categorically by making the pushout from $(i : P_T \to Q_T, f : P_T \to I_T)$. This will be proven in the following lemma:

Lemma 2.1. *Let P_T, Q_T and I_T be objects and $i : P_T \hookrightarrow Q_T$ a monomorphism and $f : P_T \to I_T$ a morphism in $\mathcal{G}_T$. The pushout of (f, i) with P_T as common object is an object I'_T and pair (i_f, f') constructed as follows:*

(1) *$I'_E = I_E \cup (Q_E - i(P_E))$ and $I'_N = I_N \cup (Q_N - i(P_N))$*[2] *and*

(2) *If $e \in I'_E$ then $s_{I'}(e)$ is computed as follows:*

- *if $e \in I_E$, then $s_{I'}(e) = s_I(e)$,*

[2] For notational convenience, we omit the indexes T of the sets of edges and nodes (X_E and X_N) of a typed graph X_T.

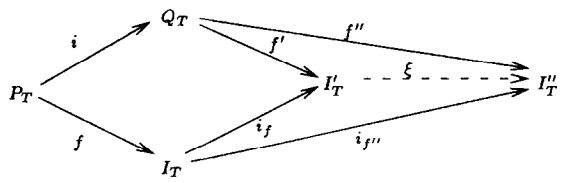


Fig. 4.

- *else if* $s_Q(e) \in Q_N - i(P_N)$, *then* $s_{I'}(e) = s_Q(e)$,
- *else* $s_{I'}(e) = f(n)$, *where* $s_Q(e) = i(n)$ *for* $n \in P_N$;

$t_{I'}$ *is defined analogously*;

(3) i_f *is the associated monomorphism from* I_T *to* I'_T; *and*

(4) $f' : Q_T \to I'_T$ *is calculated as follows:*

- *if* $e \in Q_E - i(P_E)$ *then* $f'_E(e) = e$,
- *else* $f'_E(e) = f(e^*)$, *where* $e = i(e^*)$ *for* $e^* \in P_E$.

$f'_N(n)$ *for* $n \in Q_N$ *is defined analogously.*

Proof. The proof of this lemma is completely analogous to that given in [8]. Although the lemma over there is proven for simple graphs, it can easily be seen that this construction preserves the type morphisms (in the associated category $\mathscr{G}$ to T). This is guaranteed by the fact that i is a monomorphism. □

The construction in [8] provides the foundation of the so-called *direct derivations* and *graph grammars*. In [8], the subgraphs which define how to merge the information are called *interface graphs*; all emphasis is put on the *reconstruction* of these interface graphs. While this approach is needed to define general transformations of graphs, it is not necessary to define addition operators.

We now define formally how an embedding from a pattern to an abstract instance defines a new instance which will be called the *single CGOOD-extension* of the given abstract instance with respect to the given pattern. We will also introduce an extension which will contain all information added for all embeddings from the pattern to the instance. This extension will be called the *full CGOOD-extension.*

Definition 2.1. Let P_T, Q_T and I_T be objects and $i : P_T \hookrightarrow Q_T$ a monomorphism in $\mathscr{G}_T$.

(1) Let f be an arbitrary morphism from P_T to I_T in $\mathscr{G}_T$. An object I'_T from $\mathscr{G}_T$ is called a *single CGOOD-extension* from I_T with respect to P_T and Q_T and denoted $SCGExt(P_T \overset{i}{\hookrightarrow} Q_T, P_T \overset{f}{\longrightarrow} I_T)$, if following conditions are satisfied:

- there are morphisms $i_f : I_T \hookrightarrow I'_T$ and $f' : Q_T \to I'_T$ such that $i_f \circ f = f' \circ i$; and
- if there is another object I''_T with associated morphisms f'' and i''_f satisfying the condition above, there exists a unique $\xi : I'_T \to I''_T$ making the diagram in Fig. 4 commute.

(2) An object $I_{T,G}$ is called the *full CGOOD-extension* of I_T with respect to

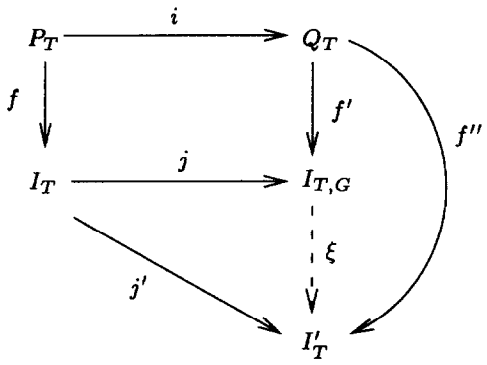


Fig. 5.

$i : P_T \hookrightarrow Q_T$ if following conditions are satisfied:

- there exists a $j : I_T \hookrightarrow I_{T,G}$ such that for each $f : P_T \to I_T$, there exists a $f' : Q_T \to I_{T,G}$ such that $f' \circ i = j \circ f$; and
- if I'_T is another object subject to the condition above (where j' denotes the fixed morphism as defined above, and f'' denotes the morphism corresponding to every $f : P_T \to I_T$), then there exists a unique arrow ξ from $I_{T,G}$ to I'_T such that for all morphisms f from P_T to I_T, the diagrams in Fig. 5 commute.

It should be noted that for every $f : P_T \to I_T$, we have a diagram such as the one in Fig. 5. The morphisms i, j and j' are fixed for all diagrams. The morphisms f' and f'' depend on the f under consideration. □

The single CGOOD-extension corresponds categorically to the pushout concept. The full CGOOD-extension is closely tied with the single CGOOD-extension. It can be shown, however, that, usually, full CGOOD-extensions cannot be written as single CGOOD-extensions and vice versa. All depends, of course, on the instance I_T and embedding P_T under consideration.

We now prove a theorem which shows how a full CGOOD-extension can be constructed in terms of single CGOOD-extensions. Afterwards, we show that, if such an object exists, it is essentially unique (up to isomorphism).

Theorem 2.2. *Let P_T, Q_T and I_T be objects and $i : P_T \hookrightarrow Q_T$ a morphism in $\mathcal{G}_T$. For each morphism f from P_T to I_T, let $(i_f : I_T \to I_{T,f}, f' : Q_T \to I_{T,f})$ be the $SCGExt(P_T \xrightarrow{i} Q_T, P_T \xrightarrow{f} I_T)$. Then the colimit $\coprod_{f \in \mathcal{G}_T(P_T, I_T)} (I \xhookrightarrow{i_f} I_{T,f})$ is an element of the isomorphism class of full CGOOD-extensions of I_T with respect to P_T and Q_T.*

Proof. Let U_T denote the colimit construction as described above. We first show that there exists a morphism $\xi : I_T \to U_T$, satisfying the first condition of the full CGOOD-extension.

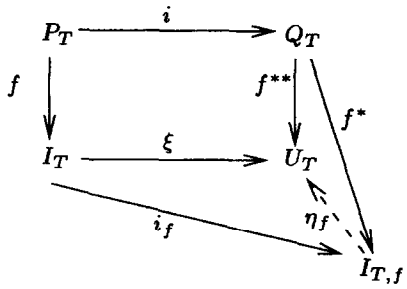


Fig. 6.

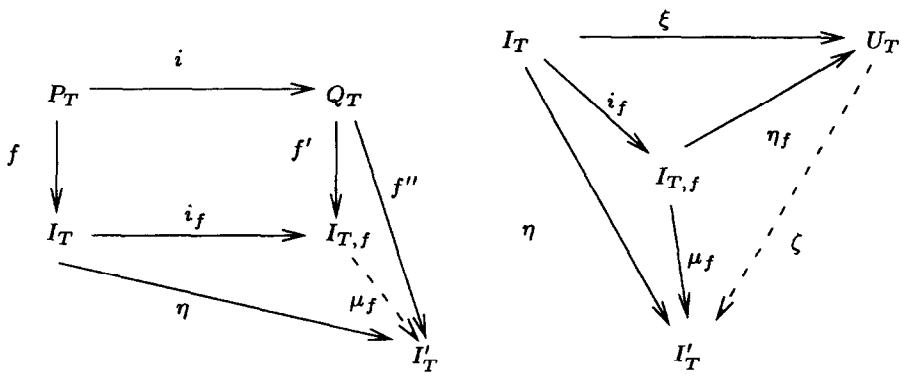


Fig. 7.

For each $f : P_T \to I_T$, we can form the pushout with $i : P_T \hookrightarrow Q_T$. This introduces an object $I_{T,f}$ and two morphisms $i_f : I_T \to I_{T,f}$ and $f^* : Q_T \to I_{T,f}$. In the category $\mathscr{G}_T$ the morphism i_f is a monomorphism too. Because U_T is the colimit of these $I_{T,f}$, there exists a series of η_f such that $\eta_f \circ i_f$ is constant. Let $\xi = \eta_f \circ i_f$. We then have $\eta_f \circ i_f \circ f = \eta_f \circ f^* \circ i$ or $\xi \circ f = f^{**} \circ i$ where $f^{**} = \eta_f \circ f^*$. The morphism ξ and the morphisms f^{**} yield the required result (see Fig. 6).

We now show that also the second condition of the full CGOOD-extension is satisfied. Let U_T be the colimit construction, and ξ the associated morphism as above. Let I'_T be another object satisfying the first condition of the full CGOOD-extension. Then, there exists a $\eta : I_T \hookrightarrow I'_T$ such that for all $f : P_T \to I_T$, there exists a $f'' : Q_T \to I'_T$ such that $f'' \circ i = \eta \circ f$.

For each f, there exists a μ_f such that the left-hand side diagram of Fig. 7 commutes. This is due to the fact that $I_{T,f}$ is a pushout. We therefore have that $\mu_f \circ i_f = \eta$. Because U_T is the colimit of all $I_{T,f}$, there exists a unique ζ making the right-hand side diagram of Fig. 7 commute, i.e., $\zeta \circ \eta_f = \mu_f$. This implies that $\eta = \zeta \circ \xi$. The morphism η, moreover, makes the complete diagram also commute, and thus satisfies the uniqueness property of the full CGOOD-extension.

This completes the proof. □

Theorem 2.3. *A full CGOOD-extension $I_{T,G}$ of an instance I_T with respect to P_T and Q_T is unique up to isomorphism.*

Proof. This is an immediate consequence of the uniqueness property of Definition 2.1. □

After these categorical constructions, we are finally ready to define the data addition operators. First we will define the *ADD_S* (i.e., *single addition*) operator which corresponds to the single extension operator as defined above.

Definition 2.4. Let $(\mathcal{G}_T, I_T)$ be a database instance defined over the scheme T and P be a pattern over that scheme; let $f : P_T \to I_T$ be an embedding from P_T to I_T and let $Q_{T'}$ contain P_T, i.e., $P \hookrightarrow Q \to T' = P \to T \hookrightarrow T'$. The database instance $ADD\_S((\mathcal{G}_T, I_T), f : P_T \to I_T, P_T \hookrightarrow Q_{T'}, T \hookrightarrow T')$ is then defined as $(\mathcal{G}_{T'}, I'_{T'})$ where $I'_{T'}$ is obtained as follows:

- $P_{T'}$ and $I_{T'}$ are obtained through composition with the monomorphism from T to T'.
- $I'_{T'}$ is the single CGOOD-extension of $I_{T'}$ with respect to $P_{T'}$ and $Q_{T'}$.

We now will define the *ADD_F* (or *full addition*) operator which corresponds to full CGOOD-extension and thus implements the full pattern matching power.

Definition 2.3. Let $(\mathcal{G}_T, I_T)$ be a database instance defined over the scheme T and P be a pattern over that scheme; let $Q_{T'}$ be an extension of P_T, i.e., $P \hookrightarrow Q \to T' = P \to T \hookrightarrow T'$. The construct $ADD\_F((\mathcal{G}_T, I_T), P_T \hookrightarrow Q_{T'}, T \hookrightarrow T')$ is defined as the database instance $(\mathcal{G}_{T'}, I_{T'})$ where $I'_{T'}$ is obtained as follows:

- $P_{T'}$ and $I_{T'}$ are obtained through composition with the monomorphism from T to T'.
- $I'_{T'}$ is the full CGOOD-extension of $I_{T'}$ with respect to $P_{T'}$ and $Q_{T'}$.

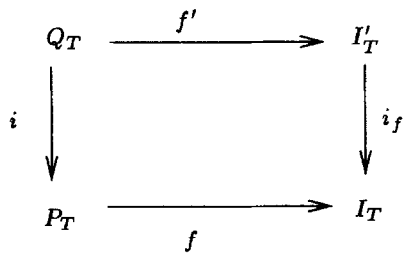
It should be noted that the addition operators are defined in a declarative, categorical way. Lemma 2.1 and Theorem 2.2, however, provide an operational way to construct both single and full CGOOD-extensions.

2.2. Deletion of information

Deletion of information will also be defined in terms of patterns and abstract instances. While the addition operators were defined in terms of the CGOOD-extension concept, the deletion operators will be based on what we will call the *CGOOD-restriction*.

The single CGOOD-restriction will be defined as the opposite of the single CGOOD-extension. Given a pattern matching in an instance and a subpattern of that pattern, the single CGOOD-restriction will determine a subgraph of the instance consisting of those data preserved in the subpattern.

Definition 2.4. Let P_T, Q_T and I_T be objects, $f : P_T \to I_T$ a morphism and $i : Q_T \hookrightarrow P_T$ a monomorphism in $\mathscr{G}_T$. An object I'_T from $\mathscr{G}_T$ is called a *single CGOOD-restriction* from I_T with respect to P_T and Q_T and denoted $SCGRest(Q_T \overset{i}{\hookrightarrow} P_T, P_T \overset{f}{\longrightarrow} I_T)$ if there exists a monomorphism $i_f : I'_T \hookrightarrow I_T$ and a morphism $f' : Q_T \to I'_T$ such that I_T is isomorphic to $SCGExt(Q_T \overset{f'}{\hookrightarrow} I'_T, Q_T \overset{i}{\longrightarrow} P_T)$.



It should be pointed out that this re-construction is not always possible. In [8], sufficient conditions are given in order to support this construction. For our purpose, however, it turns out that these conditions are usually satisfied.

Deletion of information from a database instance will also be defined for all possible embeddings from a pattern P into an instance I. For each pattern matching, the result of the single CGOOD-restriction will be retained. The global result of the deletion operator can be calculated by intersecting all subgraphs of I thus obtained. Categorically, this corresponds to the construction of a limit. This leads to following definition:

Definition 2.5. Let P_T, Q_T and I_T be objects and $i : Q_T \hookrightarrow P_T$ a monomorphism in $\mathscr{G}_T$; for each morphism $f : P_T \to I_T$, let $I_{T,f}$ be the restriction $SCGRest(Q_T \overset{i}{\hookrightarrow} P_T, P_T \overset{f}{\longrightarrow} I_T)$ with $i_f : I_{T,f} \hookrightarrow I_T$.

The *full CGOOD-restriction* of I_T with respect to P_T and Q_T is then defined as an object $I_{T,G}$ subject to following conditions:

- there exists a $f_G : Q_T \to I_{T,G}$ and, for each $f : P_T \to I_T$, a morphism $\lambda_f : I_{T,G} \to I_{T,f}$ such that $\lambda_f \circ f_G = f'$ and $i_f \circ \lambda_f = i_G$; and
- if there is another such object I'_T with associated f'_G and μ_f for each f, then there exists an unique $\xi : I'_T \to I_{T,G}$ making the associated diagrams (Fig. 8) commute.

This definition defines the CGOOD-restriction as the *largest* subobject I_G of I subject to the restriction conditions. We now can state the following theorem:

Theorem 2.4. *Let Q_T, P_T, I_T and $i_f : I_{T,f} \hookrightarrow I_T$ be defined as in the previous definition. Then the limit $\prod_{f \in \mathscr{G}_T(P_T, I_T)} (I_f \overset{i_f}{\hookrightarrow} I_T)$ is an element of the isomorphism class of full CGOOD-restrictions of I_T with respect to P_T and Q_T.*

Proof. This is an immediate consequence of the definition of the limit operator. □

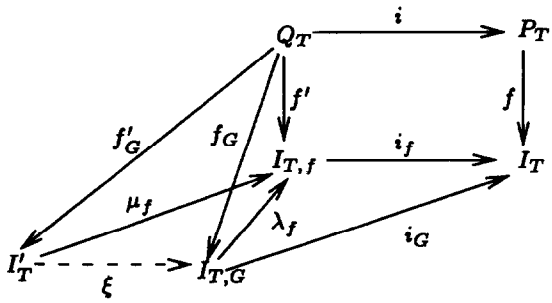


Fig. 8.

Analogous to the addition operators, we now define the two deletion operators on the database. Again, we first define a single operator (*DEL_S* or *single deletion*) which corresponds to the single CGOOD-restriction.

Definition 2.6. Let $(\mathscr{G}_T, I_T)$ be a database instance defined over the scheme T, let $f: P_T \to I_T$ be an embedding from P_T to I_T over T, and let Q_T be a subobject of P_T with $Q_T \hookrightarrow P_T$. The database instance $DEL\_S((\mathscr{G}_T, I_T), f: P_T \to I_T, Q_T \hookrightarrow P_T)$ is then defined as the database instance $(\mathscr{G}_T, I'_T)$ where I'_T is defined as $SCGRest(Q_T \overset{i}{\hookrightarrow} P_T, P_T \overset{f}{\longrightarrow} I_T)$.

The *DEL_F* (or *full deletion*) we now define corresponds to the full CGOOD-restriction operator and will delete subpatterns of patterns occurring in a database instance.

Definition 2.7. Let $(\mathscr{G}_T, I_T)$ be a database instance defined over the scheme T and let Q_T be a subobject of P_T with $Q_T \hookrightarrow P_T$. The database instance $DEL\_F((\mathscr{G}_T, I_T), Q_T \hookrightarrow P_T)$ is then defined as the full CGOOD-restriction of I_T with respect to P_T and Q_T.

2.3. Recursiveness in CGOOD

The model we introduced so far is not able to express general recursive queries. Recursiveness can be introduced in several ways. In [21], we added recursive power to the typegraph model by means of the introduction of set-types. We were able to formulate recursive queries such as the powerset and transitive closure by using the set-type constructor. As we will see in the following section, expression of the powerset is not possible in terms of the operators introduced so far.

Another approach which is used a lot in literature is the use of iteration as a tool to simulate recursive operators (cf. [10]). This technique consists of defining an operation which is carried out a number of times until the result remains unchanged, i.e., a fixpoint is reached. In CGOOD, we will take this approach.

Definition 2.8. Let $(\mathscr{G}_T, I_T)$ be a database instance defined over the scheme T; let R : $\mathscr{G}_T \to \mathscr{G}_T$ be an operator which is a concatenation of addition and deletion operators

as defined above and which associates with each instance J_T another instance $R(J_T)$ such that $J_T \hookrightarrow R(J_T)$. This introduces a chain as follows:

$$I_T \hookrightarrow R(I_T) \hookrightarrow R^2(I_T) \hookrightarrow \ldots.$$

The $ADD\_I((\mathscr{G}_T, I_T), R)$ will be defined as the colimit in $\mathscr{G}_T$ of the chain above.

In general, it will not be clear whether this colimit exists. If it exists, we will call it the *fixpoint* of the iterative addition operator.

2.4. The unique representation of data

We now have come at a point we want to reevaluate the way the data are represented in our model. As already pointed out in previous sections, the instance of a scheme is the set of typed edges which have typed sources and destinations. This approach allowed us to represent incomplete information in a natural way. A drawback of this approach, however, is the fact that there can easily be edges of the same type with the same values.

Although this poses no set-theoretic problems, one might argue that this has no meaning in a datamodel or, worse, might lead to an ambiguous representation of real world situations. From a pragmatic point of view, this objection can be dismantled by the argument that starting from database instances containing no such ambiguous arrows, the constructions defined so far will not generate any. Indeed, it can be shown that neither the colimit construction nor the applied limit construction (i.e., of monomorphisms) will introduce the ambiguous morphisms.

A more elegant argument can be given by introducing an equivalence relation $\sim_T$ on the edges.

Definition 2.9. Let G_T be an object of $\mathscr{G}_T$, i.e., defined by $s,t : E \to N$, $g_E : E \to E_T$ and $g_N : N \to N_T$. If e_1 and e_2 are elements of E, then $e_1 \sim_T e_2$, if

- $s \times t(e_1) = s \times t(e_2)$, and
- $g_E(e_1) = g_E(e_2)$.

It easily can be seen that this relation is an equivalence relation. We now consider all edges of the same equivalence class as one edge. This can be done formally by the introduction of the R_T operator and the new category $\overline{\mathscr{G}_T}$.

Definition 2.10. The category $\overline{\mathscr{G}_T}$ is defined as the subcategory of $\mathscr{G}_T$ where $Obj(\overline{\mathscr{G}_T}) = \{G_T = (s,t,g_E,g_N) \in Obj(\mathscr{G}_T) \mid \forall e_1, e_2 \in E : e_1 \sim_T e_2 \Rightarrow e_1 = e_2\}$.

This definition basically states that the category $\overline{\mathscr{G}_T}$ is the subcategory of $\mathscr{G}_T$ which only contains graphs with no ambiguous edges.

Definition 2.11. The operator $R_T : \mathcal{G}_T \to \overline{\mathcal{G}_T} \subset \mathcal{G}_T$ maps an object $G_T = (s, t, g_E, g_N)$ to an object $R_T(G_T) = (R_T(s), R_T(t), R_T(g_E), R_T(g_N))$ as follows:

- $R_T(E) = \{[e]_\sim \mid e_1 \sim_T e_2\}$ and $R_T(e) = [e]_\sim$;
- $R_T(N) = N$ and $R_T(n) = n$;
- $R_T(s) : R_T(E) \to R_T(N)$ where $R_T(s)[e] = s(e)$;
- $R_T(t) : R_T(E) \to R_T(N)$ where $R_T(t)[e] = t(e)$; and
- $R_T(g_E)$ is the restriction of g_E to $R_T(E)$ and $R_T(g_N) = g_N$.

Note that $R_T(s)[e]$ ($R_T(t)[e]$) is independent of the representative e of the equivalence class $[e]$, and thus is well-defined.

We now define how R_T transforms the morphisms of $\mathcal{G}_T$.

Definition 2.12. Let $f : G_{1,T} \to G_{2,T}$ be a morphism of $\mathcal{G}_T$. The morphism $R_T(f) : R_T(G_{1,T}) \to R_T(G_{2,T})$ is defined as follows:

- $R_T(f_E)[e] = [f_E(e)]$; and
- $R_T(f_N)(n) = [f_N(n)]$.

Theorem 2.5. *The operator $R_T : \mathcal{G}_T \to \overline{\mathcal{G}_T}$ is a forgetful functor, i.e., $R_T(id_G) = id_{R_T(G)}$ and $R_T(g \circ f) = R_T(g) \circ R_T(f)$.*

Proof. The proof is easy but we omit it here because the notation is rather involved. □

We now take a look at the behavior of the functor R_T. As it is a functor which essentially defines an equivalence relation, it can be expected that it has a right adjoint. We indeed have:

Theorem 2.6. *The functor $R_T : \mathcal{G}_T \to \overline{\mathcal{G}_T}$ as defined above has a right adjoint R'_T.*

Proof. Define $R'_T : \overline{\mathcal{G}_T} \to \mathcal{G}_T$ as the straightforward identification functor which associates with each equivalence class G_T the representative graph which has no *ambiguous* edges (i.e., edges of the same type with the same values). We now have that for any $G_{1,T} \in \mathcal{G}_T$ and $G_{2,T} \in \overline{\mathcal{G}_T}$ that $\overline{\mathcal{G}_T}(R_T(G_{1,T}), G_{2,T}) \cong \mathcal{G}_T(G_{1,T}, R'_T(G_{2,T}))$, which proves the theorem. □

Theorem 2.7. *The functor $R_T : \mathcal{G}_T \to \overline{\mathcal{G}_T}$ as defined before preserves the colimit construction.*

Proof. This is a consequence of the characterization of right adjoints. In [4] it is shown that if a functor has a right adjoint, then it preserves colimits. □

This is a very important property, because it proves that the constructions as carried out before in Lemma 2.1 to merge the information of two graphs by means of a pushout can be translated in a natural way to the new representation using equivalence

classes. This also proves that working in the category $\mathscr{G}_T$ instead of the newly defined $\overline{\mathscr{G}}_T$ poses no real problems. For notational convenience, we prefer to keep working in the category $\mathscr{G}_T$.

3. The expressive power of the language

In this section, we define a number of queries which show the expressiveness of the CGOOD language.

Thereto, we first simulate the relational database operators in CGOOD. Then, we investigate the possibility to express the so-called *abstraction* operators and their relation to the iteration construct defined in the previous section. It turns out that abstraction of functional properties can be expressed by the CGOOD model without using the iteration operator introduced in the previous section. In order to express the transitive closure, however, the iteration operator is required.

Before proceeding, we introduce a few notations to denote the different languages defined by allowing/disallowing the operators defined so far:

- $\mathscr{L}_{\mathscr{G}}^{\mathscr{S}}$: the language consisting of *ADD_S* and *DEL_S*;
- $\mathscr{L}_{\mathscr{G}}^{\mathscr{F}}$: $\mathscr{L}_{\mathscr{G}}^{\mathscr{S}}$ extended with *ADD_F* and *DEL_F*; and
- $\mathscr{L}_{\mathscr{G}}^{\mathscr{I}}$: $\mathscr{L}_{\mathscr{G}}^{\mathscr{F}}$ extended with *ADD_I*.

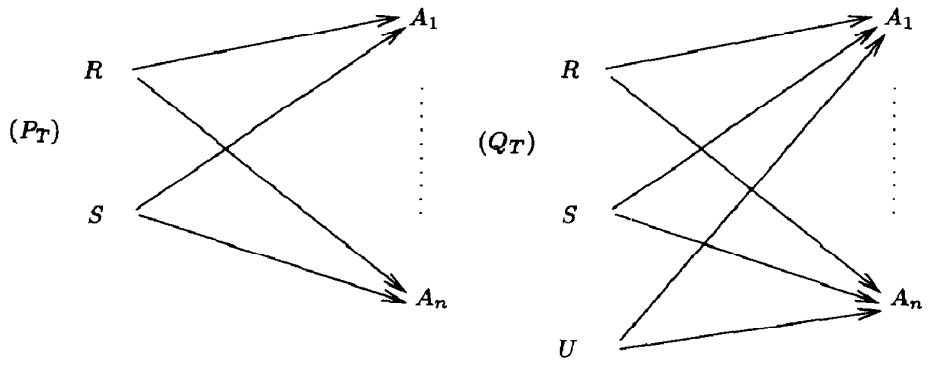
3.1. The relational database operators

In order to be able to express the relational database operators in our model, we first have to translate the relational model in a canonical way in the CGOOD formalism. This can be done easily by observing that every relation scheme R with attributes $A_1,...,A_n$ can be mapped to a typegraph T with nodes R, $A_1,...,A_n$ and edges $a_i : R \rightarrow A_i$. On instance level, there exists for every tuple $t(v_1,...,v_n)$ a node x of type R with edges of type a_i to objects x_i of type A_i.

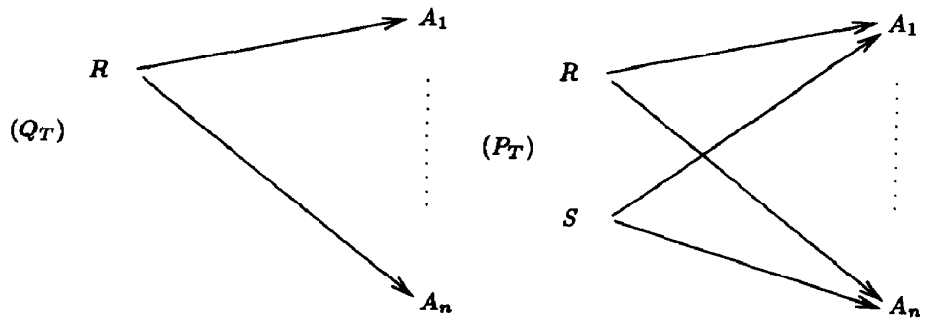
Theorem 3.1. *The relational database operators can be expressed by* $\mathscr{L}_{\mathscr{G}}^{\mathscr{F}}$.

Proof. We now express the different relational database operators in $\mathscr{L}_{\mathscr{G}}^{\mathscr{F}}$. In the relational database model, the binary operators (such as $\cup$, $\cap$ and $\bowtie$) are defined on different relations. CGOOD, however, only supports transformations within one graph. Therefore, we will introduce different relational database schemes (becoming types in CGOOD) within one CGOOD scheme.

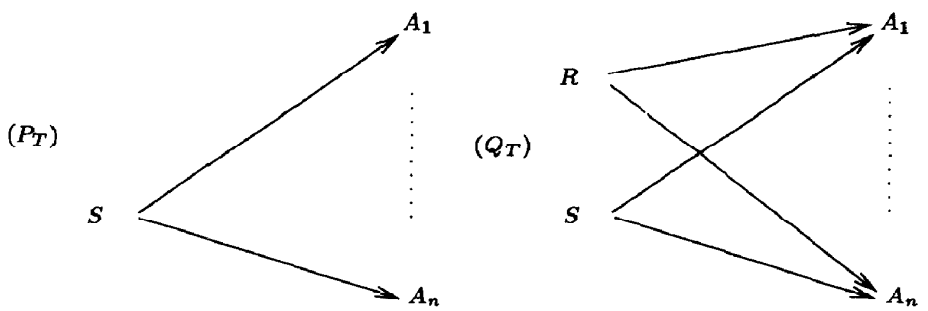
(1) *Intersection*: Let R and S be types having attributes $A_1,...,A_n$ and I_T the associated instance. Applying the *ADD_F* operator where P_T and Q_T are defined as below generates an instance $I'_{T'}$ where T' is obtained from T by adding the new type U and associated arrows to $A_1,...,A_n$. The nodes of type U represent the tuples of the intersection. The *DEL_F* operator can now be used to delete all nodes of type R and S.



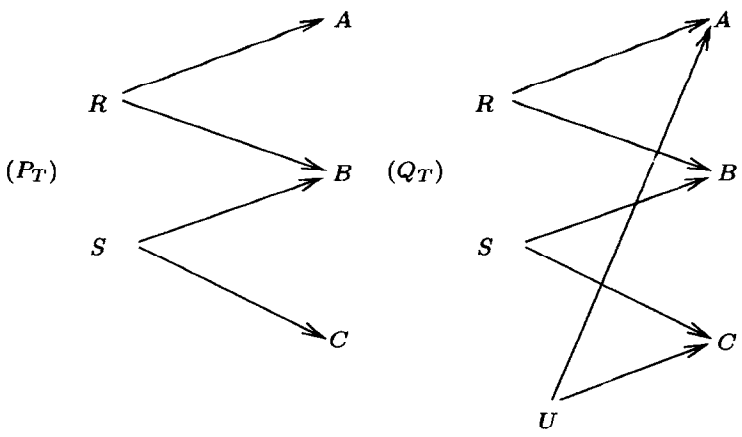
(2) *Difference*: Let R and S be defined as above. The difference operator can be simulated by using the DEL_F operator where P_T and Q_T are defined as below. Next, we delete all nodes (and morphisms) pertaining to type S. The nodes of type R yield the result.



(3) *Union*: Let R and S be as defined above. We first calculate the difference as above but preserve the nodes of type S. Then, we apply the ADD_F operator with P_T and Q_T as follows:

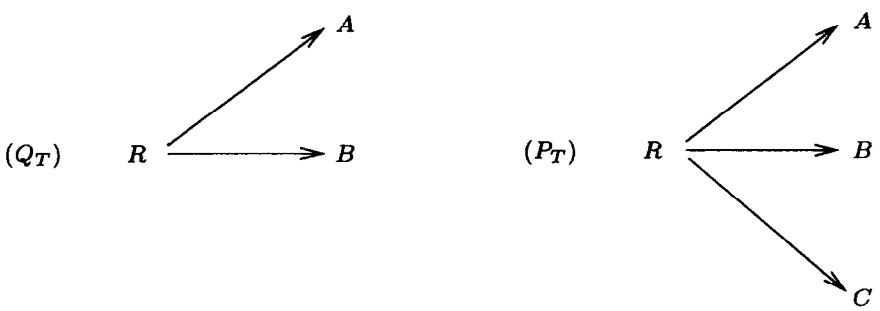


(4) *Join*: For simplicity, we assume that we have a type R with attributes A and B and a type S with attributes B and C. The joined type U with attributes A and C is obtained by applying the ADD_F operator with P_T and Q_T as follows:



This operator can introduce, however, a number of elements of type U having the same values for A and C. Therefore, we have to remove the duplicates. This is a process which is called abstraction. In the next section, we will prove that this type of simple abstraction can be expressed using the $\mathscr{L}_{\mathscr{G}}^{\mathscr{F}}$ operators.

(5) *Projection*: Assume we have a type R with attributes A, B and C. We now want to calculate the projection of R on A and B. Thereto, we first apply the DEL_F operator with P_T and Q_T as below:



The removal of the edges to type C can introduce, again, a number of elements of type R having the same values for A and B. The same remark as for the join is applicable here.

(6) *Selection*: To express selection, we have to distinguish between the equality and inequality operator. The equality operator can easily be expressed using the usual pattern matching operators. To express the inequality operator in formulas, it is not sufficient to have a pattern P_T with two different objects, because those objects can be

mapped together by the embedding into the instance. Therefore, we now show how we can add edges (of a new type) distinguishing between objects.

Let A be a type the objects of which we want to distinguish. We proceed in two steps:

- We first connect all objects of type A by introducing a new edge type a. This can be achieved by executing the ADD_F operator where P_T and Q_T are as follows:

- We now remove all edges of type a being loops, by executing the DEL_F operator and taking Q_T and P_T as follows:

By the addition of the new type a, inequality is reduced to pattern matching, and thus expressible in $\mathscr{L}_{\mathscr{G}}^{\mathscr{F}}$. □

3.2. Abstraction

In an object-oriented database model, objects may not be distinguishable on the basis of their properties because objects have an identity which makes them unique, unlike in value-based systems (cf. the relational database model, abstract data typing) where objects are solely determined by the values of their attributes.

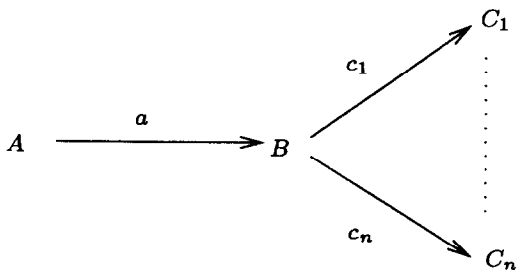
In some applications however, it is necessary to *group* those objects having the same properties. This operation is called *abstraction*. As seen before, we need an operator of this kind to express, e.g., the projection of the relational database model. We now give the formal definition of the abstraction operator in CGOOD, which is equivalent with its GOOD counterpart [9].

Definition 3.1. Let $(\mathscr{G}_T, I_T)$ be a database instance defined over the scheme T; let T contain the nodes B and $C_1, ..., C_n$ and corresponding edges $c_i: B \to C_i$, $i = 1, ..., n$. Let x be an object in I_T of type B. The *value-set* $\mathscr{V}(x, \{c_1, ..., c_n\})$ is defined as the sequence $(E_i)_i$ where $E_i = \{e \in E \mid I_T(e) = c_i \wedge s(e) = x\}$.

Definition 3.2. Let $(\mathscr{G}_T, I_T)$ be a database instance defined over the scheme T; let T contain the nodes B and $C_1, ..., C_n$ and corresponding edges $c_i : B \to C_i, i = 1, ..., n$. The *abstraction* of B with respect to the properties $c_1, ..., c_n$ of $(\mathscr{G}_T, I_T)$ is a database

instance $(\mathscr{G}'_{T'}, I'_{T'})$, where:

- T' is obtained from T by adding a node A and edge $a : A \to B$;
- I' is obtained from I by adding objects x of type A for every different value-set $\mathscr{V}(y, \{c_1, ..., c_n\})$ of objects y of type B, and edges a from x to every such y.



Definition 3.3. Let $(\mathscr{G}_T, I_T)$ be a database instance defined over the scheme T containing $t_A : R \to A$. The edge t_A of the graph T is called *functional* with respect to the instance I_T if there are no two different edges of type t_A with the same source. If an edge is not functional, it will be called *multivalued.*

It should be noted that we define the notions *functional* and *multivalued* on instance level whereas in other models (such as the relational database model) it is defined on scheme level for every possible instance. This can be explained by the fact that in our model these notions are more considered as a property than as a constraint. We only use the terminology to define the different cases of the abstraction operator.

Although one might argue that the abstraction operator as defined before is not based on categorical notions, we now will prove that the abstraction with functional links can be expressed using the full addition and deletion operators (which are defined in terms of categorical constructs).

Theorem 3.2. *The abstraction with respect to functional links can be expressed in* $\mathscr{L}^{\mathscr{F}}_{\mathscr{G}}$.

Proof. We first prove that abstraction with respect to one functional link is expressible in $\mathscr{L}^{\mathscr{F}}_{\mathscr{G}}$.

Let $(\mathscr{G}_T, I_T)$ be a database instance where T the typegraph containing $a : R \to A$. Let a be functional with respect to I_T. We now want to make the abstraction with respect to a. This can be accomplished in seven steps:

(1) We first remove all nodes of type R from which an edge of type a is leaving using DEL_F with $P_T = \{a : R \to A\}$ and $Q_T = \emptyset$. This gives an instance $I^1_T \hookrightarrow I_T$.

(2) We now delete all edges of type a and nodes of type A from I^1_T. This can be done by DEL_F with $P_T = \{a : . \to A\}$ and $Q_T = \emptyset$. This gives an instance $I^2_T \hookrightarrow I^1_T$.

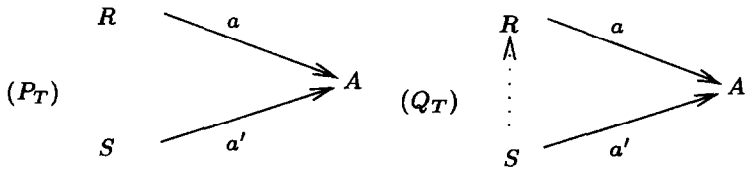
(3) We now add a single object of type A to I_T^2 (using ADD_S) giving I_T^3 with $I_T^2 \hookrightarrow I_T^3$.

(4) We now apply ADD_F to I_T^3 with $P_T = \{R, A\}$ and $Q_T = \{a : R \to A\}$. This gives an instance I_T^4 with $I_T^3 \hookrightarrow I_T^4$.

(5) We now add I_T and I_T^4 together over I_T^2 using ADD_S. This gives an instance I_T^5. (Note that I_T^5 is nothing else but the extension of I_T where we added an a-link to a new object of type A from each object of type R which had no a-link in the first place.)

(6) We now generate for each object of type a an object of a new type S and new edge of type a', using ADD_F where $P_T = \{A\}$ and $Q_T = \{a' : S \to A\}$. This gives an instance $I_{T'}^6$ where T' is obtained from T by adding the node S and edge a'.

(7) The abstraction from R over a is now obtained by adding the required edges from type S to type R by executing ADD_F with P_T and Q_T as below.



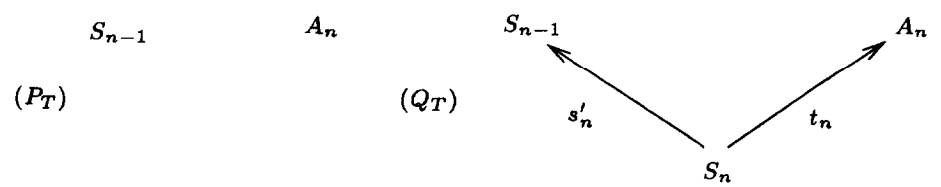
This shows the first part of our proof.

We now use induction to show that abstraction over multiple functional links can also be expressed using the $\mathscr{L}_{\mathscr{G}}^{\mathscr{F}}$ operators.

Let $(\mathscr{G}_T, I_T)$ be a database instance over a scheme T containing $a_i : R \to A_n$ and (by induction hypothesis) let $s_{n-1} : S_{n-1} \to R$ be the abstraction with respect to $a_1, ..., a_{n-1}$. We now calculate the abstraction with respect to $a_1, ..., a_n$. Thereto, we proceed as follows:

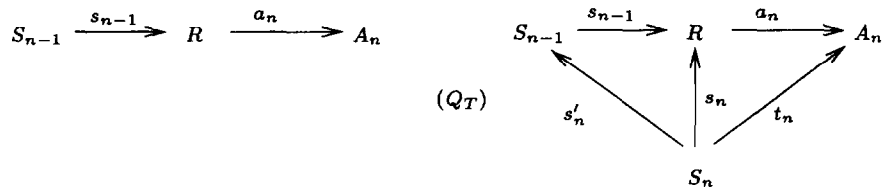
(1) The first step consists of repeating the construction to add *null*-links as before for all nodes of type R from which no edge of type a_n is leaving. This can be accomplished by repeating steps 1 to 7 as before. This gives a new instance I_T^1.

(2) We now execute the ADD_F operator on I_T^1 with P_T and Q_T defined as follows:



This gives an instance $I_{T'}^1$ with T' obtained from T by adding a type S_n and edges $s'_n : S_n \to S_{n-1}$ and $t_n : S_n \to A_n$.

(3) The abstraction $s_n : S_n \to R$ is obtained by applying ADD_F on $I^1_{T'}$ giving $I^2_{T'}$, where P_T and Q_T are defined as follows:



The unnecessary edges now can be removed using DEL_F. □

In [22], it has been shown that the general abstraction as defined above cannot be expressed in a framework which supports operators as those introduced so far. Another result from [22] says that extending a language with the powerset construction adds exactly the same power as adding the general abstraction. Even the availability of a fixpoint operator is not sufficient to express general powerset (or abstraction) because it is not guaranteed to stop. When finitary transformations are considered however, this argument is no longer valid (as can be found in [5, p. 291]).

3.3. Transitive closure

Transitive closure is an operator which typically can be expressed using the iteration operator. This is also the case in our model.

Theorem 3.3. *The transitive closure can be expressed in* $\mathscr{L}^{\mathscr{I}}_{\mathscr{G}}$.

Proof. Let $(\mathscr{G}_T, I_T)$ be a database instance over a scheme T containing types R, A and edges *from*, *to* : $R \to A$. We now want to calculate the transitive closure over *from* and *to*, which comes down to enumerating those pairs of type A which can be reached by composing the *from, to* pairs.

We first introduce a new type S and two edges $from_S$ and to_S (resulting in a new typegraph T') and copy all objects (edges) of type R (*from*, *to*) to objects (edges) of type S ($from_S$, to_S). This introduces an instance $I'_{T'}$.

Then, we define the operator over which we will apply the iteration operator.

- Apply one transitive step over S by executing the ADD_F operator where P_T and Q_T are defined as in Fig. 9.
- Apply abstraction with respect to S over the functional links $from_S$ and to_S. This introduces a new type U.
- Rename the type U to S. (This can be achieved doing the necessary copy and delete instructions.)

This iteration gives an instance $I''_{T'}$; the objects of type S denote the required pairs of the transitive closure. □

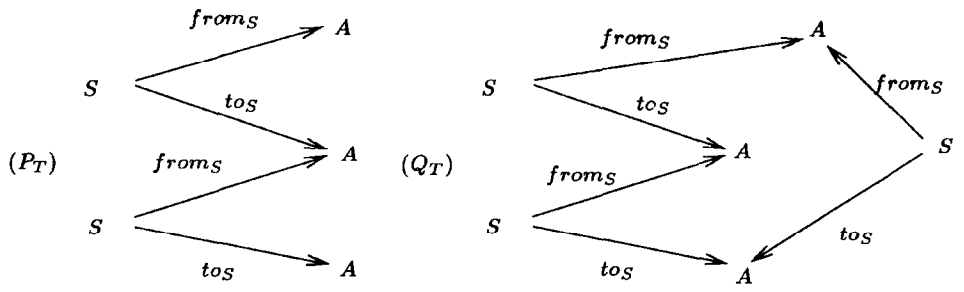


Fig. 9.

4. Conclusions and directions for future research

In this article, we presented a powerful graph-based object-oriented data model which was formulated entirely in terms of graph categories. Instances in our model were typed graphs; between these typed instance graphs, we defined morphisms being the classical pattern-matching embeddings.

In addition, we defined a number of operators capable of adding and deleting nodes and edges to instances. These operators required a pattern graph as input parameter indicating where the modifications needed to take place. A second parameter denoted what modification needed to be done. This allowed to formulate powerful operations based on the occurrence of patterns (of arbitary complexity) in the instance graph.

We proved that the CGOOD language is capable of expressing the relational database operators as well as the functional abstraction. In order to model recursive queries such as the transitive closure, however, the addition of a special fixpoint operator was required. It also became clear that using category theory as a framework for data modeling has many advantages such as offering declarative definitions with immediate operational equivalents, simplifying proofs, and offering new insights in existing concepts.

In related papers, we further investigate how category theory can be used for data modeling purposes. We already initiated work in this respect by introducing a general category based on views, which is capable of characterizing data models on a higher *meta*-level. The fundamental character of this approach has already been shown by the fact that both value-based and object-oriented data models can be modeled elegantly in this view-based framework. Among other results, this approach relates many theorems from decomposition theory to object identity. For further details, we refer to [19].

The view-based framework also allows to characterize various properties of data models independent of a specific data model. Very specific properties (such as, e.g., object identity, inheritance, data encapsulation and subclassing) can thus be defined in a universal framework and studied from a more fundamental perspective by expressing them in different contexts [18, 20].

References

[1] S. Abiteboul and C. Beeri, On the power of languages for complex objects, INRIA Tech. Report No. 806, 1988.

[2] M.A. Arbib and E.G. Manes, *Arrows, Structures and Functors* (Academic Press, New York, 1975).

[3] F. Bancilhon and S. Khoshafian, A calculus for complex objects, in: *Proc. 4th PODS*, Portland, 1985.

[4] M. Barr and C. Wells, *Toposes, Triples and Theories*, Grundlehren der mathematischen Wissenschaften, Vol. 278 (Springer, Berlin, 1985).

[5] M. Barr and C. Wells, *Category Theory for Computing Science* (Prentice-Hall, Englewood Cliffs, NJ, 1990).

[6] J.L. Bell, *Categories, Toposes and Sets* (Reidel, Dordrecht, 1982).

[7] E.F. Codd, A relational model of data for large shared data banks, *Comm. ACM* **13** (1970) 377–387.

[8] H. Ehrig, Introduction to the algebraic theory of graph grammars, Lecture Notes in Computer Science, Vol. 73 (Springer, Berlin, 1978) 1–69.

[9] M. Gyssens and J. Paredaens, A graph-oriented object model for database end-user interfaces, in: *Proc. ACM SIGMOD Internat. Conf. on Management of Data*, Atlantic City, 1990.

[10] M. Gyssens and D. Van Gucht, Powerset algebra as a result of adding programming constructs to the nested relational algebra, in: *Proc. ACM SIGMOD Internat. Conf. on Management of Data*, Chicago (1988) 225–232.

[11] H. Herrligh and G.E. Strecker, *Category Theory* (Allyn and Bacon, Boston, 1973).

[12] R. Hull and J. Su, On accessing object-oriented databases: expressive power, complexity and restrictions, in: *Proc. ACM SIGMOD Internat. Conf. on Management of Data*, 1989.

[13] M. Kifer and J. Wu, A logic for object-oriented logic programming (Maier's O-logic Revisited), in: *Proc. 8th PODS*, Philadelphia (1989) 379–393.

[14] Z.M. Özsoyoğlu and L.-Y. Yuan, On the normalization in nested relational databases, in: S. Abiteboul, P.C. Fischer, H.-J. Schek Eds., *Nested Relations and Complex Objects in Databases*, Lecture Notes in Computer Science, Vol. 361 (Springer, Berlin, 1989) 243–271.

[15] J. Paredaens, P. De Bra, M. Gyssens and D. Van Gucht, The structure of the relational database model, EATCS Monographs on Theoretical Computer Science, Vol. 17 (Springer, Berlin, 1989).

[16] A. Siebes, On complex objects, Ph.D. Thesis at the University of Twente, 1990.

[17] S.J. Thomas, P.C. Fischer, Nested relational structures, in: P.C. Kanellakis, ed., *The Theory of Databases, Advances in Computer Research III* (JAI Press, Greenwich, CT, 1986) 269–307.

[18] C. Tuijn, Data modeling from a categorical perspective, Ph.D. Thesis at the University of Antwerp, 1994.

[19] C. Tuijn and M. Gyssens, Views and decompositions from a categorical perspective, in: *Proc. 4th Internat. Conf. on Database Theory*, Lecture Notes in Computer Science, Vol. 646 (Springer, Berlin, 1992) 99–112.

[20] C. Tuijn and M. Gyssens, A categorical approach to fundamental properties of object-oriented data models, Internal UIA Report, Antwerp, 1993.

[21] C. Tuijn, M. Gyssens and J. Paredaens, A categorical approach to object-oriented data modeling, in: *Proc. 3rd Workshop on Foundations of Models and Languages for Data and Objects*, Aigen (1991) 187–196.

[22] J. Van den Bussche and J. Paredaens, The expressive power of structured values in pure OODB's, in: *Proc. 10th ACM SIGMOD-SIGACT-SIGART Symp. on Principles of Database Systems* (1991) 291–299.